Application Note

How to Integrate Linux Driver Into Your System

Alvaro Reyes

ABSTRACT

Linux drivers are essential software components that allow the operating system to communicate with hardware devices such as graphics cards, printers, and Ethernet physical layer devices (PHY). Without the drivers, Linux will be unable to use the hardware effectively, resulting in devices not being recognized or functioning properly. This document aims to provide comprehensive guidance for developers seeking to integrate PHY functionality into Linux-based systems. By detailing the intricacies of PHY driver implementation within the Linux kernel, this application note equips developers with the knowledge and tools necessary to make sure seamless integration, designed for performance, and compatibility across a wide range of hardware platforms.

Table of Contents

Trademarks

1 Texas Instruments Ethernet PHY Drivers

All Ethernet drivers, RTOS and Linux, can be found at TI's Ethernet software github.

Table 1-1. TI Ethernet PHY Drivers

Driver	Supported Devices	U-boot Driver	Linux Driver
dp83822.c	DP83822	N/A	Yes, all can be found here: TI's Ethernet software github
	DP83825	N/A	
	DP83826	N/A	
dp83848.c	DP83848	N/A	
	DP83620	N/A	
dp83867.c	DP83867	Yes	
dp83869.c	DP83869	Yes	
dp83tc811.c	DP83TC812	N/A	
dp83tc812.c	DP83TC812	N/A	
	DP83TC813	N/A	
	DP83TC814	N/A	
dp83tg720.c	DP83TG720	N/A	
	DP83TG721	N/A	

2 Ethernet PHY Driver Overview

Ethernet PHY Linux drivers play a crucial role in enabling communication between the network interface controller (NIC) and the physical Ethernet medium. The drivers interact with the Linux kernel's networking subsystem, providing a standardized interface for high-level networking protocols and applications. Implementing Ethernet PHY drivers involves handling tasks such as auto-negotiation, link detection, speed and duplex configuration, and error handling. Additionally, these drivers often support various Ethernet standards, including 10/100/1000 Mbps Ethernet.

Figure 2-1 serves as an example that illustrates the role of an Ethernet PHY driver. Starting from the top, the user inputs a command through the Terminal (an ethtool command, for example). Ethtool, a Linux network utility, takes the input provided by the user in the terminal and checks if the parameters given are valid. This is an important step that provides a high-level interface for the user to interact with the kernel, without having direct Kernel control. If the parameters are correct, ethtool passes the command to the MAC and PHY drivers. The drivers have function definitions to execute the commands that were originally provided by the user and apply them to the hardware.

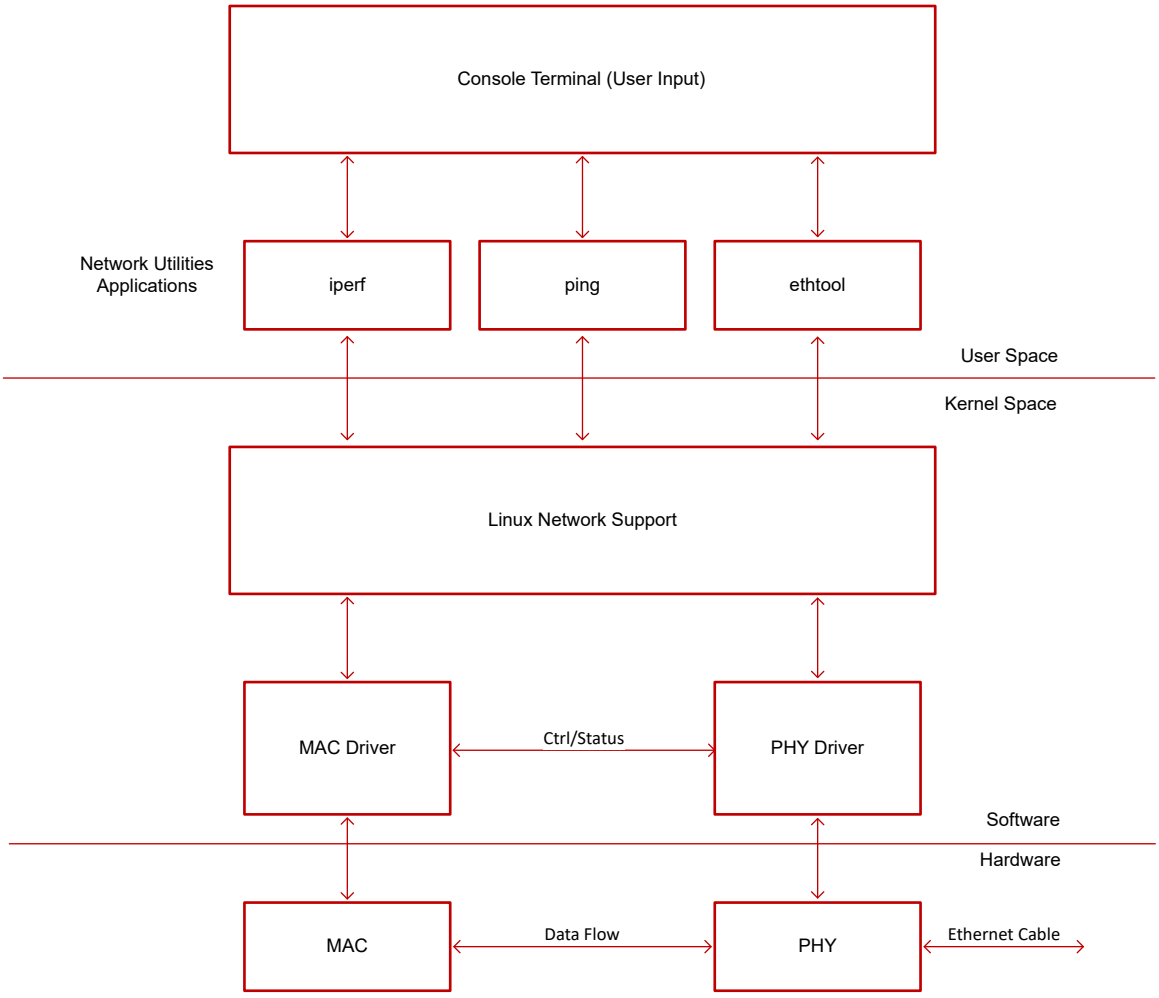

Figure 2-1. Linux Driver Block Diagram

2.1 Exploring Linux Driver Types

Within the Linux operating system, there are two main driver types that are discussed in this application note: Kernel and U-boot.

2.1.1 U-Boot Driver

A U-boot driver is a software component designed specifically for the U-Boot bootloader, which is commonly used in embedded systems and bootstrapping processes. These drivers enable U-Boot to interact with and control various hardware peripherals during the boot-up sequence. This provides proper system initialization and hardware setup before handing over control to the operating system kernel.

2.1.2 Kernel Driver

A kernel driver, also known simply as *driver*, is a software component that enables the operating system kernel to communicate with and control hardware devices. These drivers are integrated into the kernel and are responsible for managing the interactions between software applications and hardware peripherals such as network adapters, storage drives, input or output devices, and more. Kernel drivers handle tasks such as device initialization, data transfer, interrupt handling, power management, and error handling.

3 Driver Integration

Integrating a driver into a Linux system involves several key steps to provide seamless compatibility and functionality. Initially, developers must compile the driver code to generate a loadable kernel module or incorporate the code directly into the kernel (the latter is preferred for faster PHY recognition during MDIO probe).

The following sections describe the process in greater detail with the following setup. The J721EXCPXEVM common processor board is used with the J721EXSOMG01EVM TDA4VM and DRA829V socketed system on module. The Linux-RT SDK is used to evaluate Linux Kernel version 5.10 on the board. The common processor board natively has one Ethernet port, using the DP83867E Ethernet PHY. A plug in daughter card with four additional Ethernet ports is plugged into the common processor board's EVM expansion connector, where the drivers for the Ethernet PHYs are not included in the processor's SDK.

3.1 Linux Device Tree

A Linux device tree is a data structure used to describe the hardware components and configuration in embedded systems. The device tree provides a standardized way for the operating system to understand the hardware layout, including details about the processor, memory, buses. and peripherals. Device tree data is typically stored in a binary format (**.dtb** file) and is passed to the Linux kernel during boot-up. The kernel then uses this information to dynamically bind device tree nodes and initialize hardware components, allowing for efficient and flexible hardware support across different embedded platforms without the need for hard-coding hardware details into the kernel.

Device Tree Code block is an example of how four Ethernet PHYs on the daughter card are configured in the device tree file. CPSW refers to the MAC interface of the processor and the main node definitions to consider are:

- &cpsw0 {} which initializes four RGMII interfaces
- cpsw0_port*n* {} which initializes further details for each port
  - phy-mode: sets the MAC interface of that port
  - phy-handle: defines how to set up the PHY
    - <&cpsw9g_phy*x*> is used to set the PHY Address
      - Note that *x* does not set the PHY Address and is only a naming convention. The address is assigned lower in the codeblock inside the cpsw9g_mdio{} definition.
        - reg = <*x*>;
    - The RGMII delays can be set here too, an example can be seen in the J721E common processor board dts file, line 744, and in the RGMII Codeblock.
    -
```
&davinci_mdio {
    phy0: ethernet-phy@0 { //PHY0 is defined and passed to phy-handle
        reg = <0>;
        ti,rx-internal-delay = <DP83867_RGMIIDCTL_2_00_NS>;
        ti,fifo-depth = <DP83867_PHYCR_FIFO_DEPTH_4_B_NIB>;
    };
};

&cpsw_port1 {
    phy-mode = "rgmii-rxid";
    phy-handle = <&phy0>;
};
```
      - Typically our RGMII delay recommendation is to configure the PHY to delay both TX and RX CLK by 2.0ns (referred to as *shift mode*), while the processor is set to 0 delay (referred to as *align mode*). See Table 3-1 for more information.
        -

Table 3-1. RGMII Shift Configurations

MAC Configuration	Required PHY Configuration
Align on RX	Shift on RX
Shift on RX	Align on RX
Align on TX	Shift on TX
Shift on TX	Align on TX

- However, many TI processors have an internal 2.0ns delay on the TX lines that cannot be disabled. In the RGMII Codeblock example above, only RX delay is configured on the PHY for this reason.

– phys
  - Setting which eth# each port is assigned
    – <&cpsw0_phy_gmii_sel *n*>

```
&cpsw0 {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&mdio_pins_default
             &rgmii1_pins_default
             &rgmii2_pins_default
             &rgmii3_pins_default
             &rgmii4_pins_default
             >;
};

&cpsw0_port1 {
    phy-handle = <&cpsw9g_phy0>;
    phy-mode = "rgmii-rxid";
    mac-address = [00 00 00 00 00 00];
    phys = <&cpsw0_phy_gmii_sel 1>;
};

&cpsw0_port2 {
    phy-handle = <&cpsw9g_phy4>;
    phy-mode = "rgmii-rxid";
    mac-address = [00 00 00 00 00 00];
    phys = <&cpsw0_phy_gmii_sel 2>;
};

&cpsw0_port3 {
    phy-handle = <&cpsw9g_phy5>;
    phy-mode = "rgmii-rxid";
    mac-address = [00 00 00 00 00 00];
    phys = <&cpsw0_phy_gmii_sel 3>;
};

&cpsw0_port4 {
    phy-handle = <&cpsw9g_phy8>;
    phy-mode = "rgmii-rxid";
    mac-address = [00 00 00 00 00 00];
    phys = <&cpsw0_phy_gmii_sel 4>;
};
&cpsw9g_mdio {
    bus_freq = <1000000>;
    #address-cells = <1>;
    #size-cells = <0>;

    cpsw9g_phy0: ethernet-phy@0 {
        reg = <0>;
    };
    cpsw9g_phy4: ethernet-phy@4 {
        reg = <4>;
    };
    cpsw9g_phy5: ethernet-phy@5 {
        reg = <5>;
    };
    cpsw9g_phy8: ethernet-phy@8 {
        reg = <8>;
```

```
    };
};
```

When the board is running, the terminal command *dmesg grep | mdio* can be used to confirm the PHY address (phy[*x*]) and eth port (eth*n*).

```
davinci_mdio c000f00.mdio: phy[0]: device c000f00.mdio:00, driver TI DP83TG720CS1.1
davinci_mdio c000f00.mdio: phy[4]: device c000f00.mdio:04, driver TI DP83TG721CS1.0
davinci_mdio c000f00.mdio: phy[5]: device c000f00.mdio:05, driver TI DP83TC812CS2.0
davinci_mdio c000f00.mdio: phy[8]: device c000f00.mdio:08, driver TI DP83TC814CS2.0
am65-cpsw-nuss c000000.ethernet eth4: PHY [c000f00.mdio:08] driver [TI DP83TC814CS2.0] (irq=POLL)
am65-cpsw-nuss c000000.ethernet eth3: PHY [c000f00.mdio:05] driver [TI DP83TC812CS2.0] (irq=POLL)
am65-cpsw-nuss c000000.ethernet eth2: PHY [c000f00.mdio:04] driver [TI DP83TG721CS1.0] (irq=POLL)
am65-cpsw-nuss c000000.ethernet eth1: PHY [c000f00.mdio:00] driver [TI DP83TG720CS1.1] (irq=POLL)
```

3.2 Integrating Driver

This section describes how to add a driver (newDriver.c, where newDriver is an Ethernet PHY) to an SDK on a linux system that is either missing the driver or using an outdated version.

In the SDK, find the Linux kernel directory (LKD). An example file path looks like:

```
SDK_Install_Directory/board-support/TI-linux-kernel/
```

TI-Linux-kernel is the LKD in this example. From here, you can navigate to:

```
LKD/drivers/net/phy/
```

Copy newDriver.c into this directory. Within this same directory are *Makefile* and *Kconfig* , both files need to be edited for newDriver.c to be built.

Edit Makefile

Add the following line to the Makefile. Note the assignment is newDriver.o and not newDriver.c

```
obj-$(CONFIG_newDriver_PHY)    += newDriver.o
```

Edit Kconfig

Add the following lines to the Kconfig,

```
config newDriver PHY
    tristate "<Insert Company name> newDriver PHY"
    --help--
        Supports the newDriver PHY.
```

After both the Makefile and Kconfig files have been edited, return to the LKD. From here, go to:

```
LKD/arch/arm64/configs
```

Note

If your processor is 32 bit instead of 64 bit, go into the 'arm' folder instead of 'arm64'.

Here you can find a defconfig file, add the following line:

```
CONFIG_newDriver_PHY = y
```

The naming convention, *CONFIG_newDriver_PHY*, needs to match what was set in the Makefile.

From here, you can return to the SDK install directory and run the *make* command on the terminal.

Note

Not all kernel's can be built by running *make*, consult your SDK's documentation for correct procedure to build kernel, u-boot, and dtb files.

4 Common Terminal Commands

The following are several commands accompanied with descriptions, use cases, and terminal outputs.

4.1 Initialization Commands

After system bootup, the following commands state whether the driver was loaded correctly or not.

4.1.1 dmesg | grep -i mdio

Dmesg is a Linux command that displays messages written to the kernel. The **|** symbol is known as the pipe command, which connects the output of one command directly into the input of another. Grep is a Linux command used to find strings and the **-i** parameter ignores the case of the string. Overall, **dmesg | grep -i mdio** finds all messages written to the kernel and filters for those containing *mdio*. The MDIO interface is how the processor can access the PHY's registers.

The purpose of this command is to confirm if the driver is loaded correctly or provide several debug clues on what is causing the PHY to misbehave from a software standpoint.

Examples of bad output:

```
davinci_mdio c000f00.mdio: phy[10]: device c000f00.mdio:0a, MDIO device at address 10 is missing.
```

This message indicates that the PHY is not found on the MDIO bus, which can be caused by several issues. The most common being a missing or incorrect device tree (see Section 3.1 for more information), but can also be due to a non-functional PHY or a bad MDIO connection.

Once the PHY can be detected on the MDIO bus, another common error message is:

```
am65-cpsw-nuss c000000.ethernet eth1: PHY [c000f00.mdio:0a] driver [Generic PHY] (irq=POLL)
davinci_mdio c000f00.mdio: phy[10]: device c000f00.mdio:0a, driver unknown
```

Both the *driver unknown* and *Generic PHY* messages indicate that the driver file is not loaded correctly, built, or completely missing; and Linux loaded a generic driver that won't function well with the PHY. In this case, verify that the driver was successfully compiled and added to Linux. See Section 3.2 for more information on this process.

Finally, an example of a good output looks like this:

```
root@j7-evm:~# dmesg | grep mdio
davinci_mdio 46000f00.mdio: phy[0]: device 46000f00.mdio:00, driver TI DP83867
am65-cpsw-nuss 46000000.ethernet eth0: PHY [46000f00.mdio:00] driver [TI DP83867] (irq=POLL)
```

Here we can see the *phy[0]* is identified as the DP83867 and assigned as port *eth0*

Note

PHY[*n*], where *n* represents the PHY Address can be different than the eth*x* where *x* represents which port that PHY is assigned to. For example, PHY address can be 8 while being assigned to port eth0.

4.1.2 ifconfig

ifconfig(case sensitive) is a Linux terminal command that displays network interfaces and can also be used to determine if the driver has been loaded correctly. **ifconfig -ethx down** deactivates the interface and **ifconfig -ethx up** activates it; which loads the driver again, similarly to when the board is powered on initially.

```
root@j7-evm:~# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500  metric 1
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:f9:5b:d7:a4  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500  metric 1
        ether 34:08:e1:59:5c:d2  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500  metric 1
        ether d2:eb:75:2d:68:21  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536  metric 1
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 82  bytes 6220 (6.0 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 82  bytes 6220 (6.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

4.2 Functional Commands

The commands listed in this section provide a service or function.

4.2.1 Phytool

Phytool is a Linux command for MDIO register access, this provides an easy way to read and write registers of a PHY. This tool is already integrated into TI SDK's but can be downloaded by entering the following *sudo apt-get install -y net-tools*.

Phytool command:

- *phytool read ethx/PHYADDRESS/desiredRegister*
- *phytool write ethx/PHYADDRESS/desiredRegister writeValue*

```
root@j7-evm:~# phytool read eth1/10/0x0    //Read Register 0x0
0x1140                                     //Result
root@j7-evm:~# phytool write eth1/10/0x0 0x0140     //Write Reg 0x0 = 0x0140
root@j7-evm:~# phytool read eth1/10/0x0             //Read again to confirm write command
0x0140
```

4.2.2 Ethtool

Ethtool is used to access or change network driver settings.

- *Ethtool -ethx*
  - Eth*x* represents which network device you are referring to. If a board has two Ethernet ports, there can be eth0 and eth1
  - Command tells current status and configuration of that Ethernet device
    - This is an easy way to confirm the PHYADDRESS of a port

```
ethtool eth0
Settings for eth0:
        Supported ports: [ TP    MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Full
        Supported pause frame use: Symmetric
        Supports auto-negotiation: Yes
        Supported FEC modes: Not reported
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Full
        Advertised pause frame use: Symmetric
        Advertised auto-negotiation: Yes
        Advertised FEC modes: Not reported
        Link partner advertised link modes:  10baseT/Half 10baseT/Full
                                             100baseT/Half 100baseT/Full
                                             1000baseT/Full
        Link partner advertised pause frame use: Symmetric Receive-only
        Link partner advertised auto-negotiation: Yes
        Link partner advertised FEC modes: Not reported
        Speed: 1000Mb/s
        Duplex: Full
        Auto-negotiation: on
        master-slave cfg: preferred slave
        master-slave status: slave
        Port: Twisted Pair
        PHYAD: 0
        Transceiver: external
        MDI-X: Unknown
        Supports Wake-on: ubgs
        Wake-on: d
        SecureOn password: 00:00:00:00:00:00
        Current message level: 0x000020f7 (8439)
                               drv probe link ifdown ifup rx_err tx_err hw
        Link detected: yes
```

4.2.3 Forced Master/Slave

This function forces the network interface into *Master/Slave*, without the user needing to know which registers to write to. This is particularly important for Single Pair Ethernet devices (SPE). In SPE communication, there needs to be a *Master* and *Slave*, both cannot be *Master* or *Slave*.

In the following codeblock, the *ethtool* command is used to check the current state of the PHY, then the state is changed to the other and checked again.

Command:

- ethtool -s eth*x* master-slave forced-master
- ethtool -s eth*x* master-slave forced-slave

Note

This command has no output. *ethtool eth3* | *grep master-slave* is run to check the current status

```
root@j7-evm:~# ethtool eth3 | grep master-slave
        master-slave cfg: forced master
        master-slave status: master
root@j7-evm:~# ethtool -s eth3 master-slave forced-slave
root@j7-evm:~# ethtool eth3 | grep master-slave
        master-slave cfg: forced slave
        master-slave status: slave
root@j7-evm:~#
```

4.3 Diagnostic Commands

The commands listed in this section are for debugging Ethernet specific application issues.

4.3.1 SQI

Signal Quality Indicator (SQI), a feature that is not implemented in all PHYs, is used to check the quality of the signal (**0** being no link and **7** the best possible). Useful for debugging the origin of errors, for example: link is stable but the system is seeing packet loss. A low SQI value indicates the issue is likely within the MDI interface (connector side). If SQI is implemented in the device's driver, SQI can be found in the *ethtool* command. If you only want to display the SQI, you can use *ethtool ethx | grep SQI*.

```
ethtool eth3
Settings for eth3:
        Supported ports: [ TP    MII ]
        Supported link modes:   100baseT/Full
        Supported pause frame use: Symmetric
        Supports auto-negotiation: No
        Supported FEC modes: Not reported
        Advertised link modes:  Not reported
        Advertised pause frame use: Symmetric
        Advertised auto-negotiation: No
        Advertised FEC modes: Not reported
        Speed: 100Mb/s
        Duplex: Full
        Auto-negotiation: off
        master-slave cfg: forced master
        master-slave status: master
        Port: Twisted Pair
        PHYAD: 5
        Transceiver: external
        MDI-X: Unknown
        Supports Wake-on: d
        Wake-on: d
        Current message level: 0x000020f7 (8439)
                               drv probe link ifdown ifup rx_err tx_err hw
        Link detected: yes
        SQI: 4/7
ethtool eth3 | grep SQI
        SQI: 4/7
```

4.3.2 TDR

Time Domain Reflectometer (TDR) is a function that identifies a fault in the cable. Not all Ethernet PHYs have the TDR feature, be sure to check the PHY's data sheet to confirm. For TI's automotive single pair Ethernet (SPE) PHYs to correctly run TDR, the *Master/Slave* state of the PHY must be known.

When PHY is *Master*:

- If cable is connected (good link)
  - PHY drops link, performs TDR, and regains link
- If cable is disconnected or damaged
  - PHY performs TDR and outputs:
    - Fault Type
      - Open or Short
    - Distance of fault in meters

When PHY is *Slave*:

- If cable is connected (good link)
  - Link partner (*Master*) needs to be forced silent (not transmitting any packets), otherwise TDR will fail
    - Cable can be disconnected from *Master* to run TDR as *Slave*

In the eth3 codeblock, *eth3* is initially linked up with a known good cable and configured as *Master*. TDR is run and completes as expected, with no fault detected. After TDR is completed, the cable is unplugged from the link partner, resulting in *eth3: Link is Down*. TDR is then run again.

TDR command: *ethtool --cable-test ethx*

```
root@j7-evm:~# ethtool --cable-test eth3
am65-cpsw-nuss c000000.ethernet eth3: Link is Down
PHY is set as Master.
Cable test started for device eth3.
Cable test completed for device eth3.
Pair A code OK
TDR HAS COMPLETED AND PASSED
root@j7-evm:~# No Fault Detected.
am65-cpsw-nuss c000000.ethernet eth3: Link is Up - 100Mbps/Full - flow control off

am65-cpsw-nuss c000000.ethernet eth3: Link is Down

root@j7-evm:~# ethtool --cable-test eth3
PHY is set as Master.
Cable test started for device eth3.
Cable test completed for device eth3.
Pair A code Open Circuit
TDR HAS COMPLETED AND PASSED
Open Cable Detected
Length of Fault: 3 Meters
```

In the eth4 codeblock, *eth4* is initially linked up with a known good cable and configured as *Slave*. TDR is run and fails as expected. Next the cable is unplugged from the link partner, resulting in *eth4: Link is Down*. TDR is then run again.

```
root@j7-evm:~# ethtool --cable-test eth4
am65-cpsw-nuss c000000.ethernet eth4: Link is Down
PHY is set as Slave.
Cable test started for device eth4.
Cable test completed for device eth4.
TDR HAS FAILED
root@j7-evm:~# am65-cpsw-nuss c000000.ethernet eth4: Link is Up - 100Mbps/Full - flow control off

am65-cpsw-nuss c000000.ethernet eth4: Link is Down

root@j7-evm:~# ethtool --cable-test eth4
PHY is set as Slave.
Cable test started for device eth4.
Cable test completed for device eth4.
Open Circuit
TDR HAS COMPLETED AND PASSED
Open Cable Detected
Length of Fault: 3 Meters
```

4.3.3 Throughput Testing - Ping and iPerf

Throughput testing refers to sending data from one board to another. An initial *ping* test can be performed first to confirm communication with the destination is possible. The following example demonstrates the most basic type of ping, where the host is directly connected to the destination. This example does not include any kind of switch, hub, or router.

Ping Example:A testboard running Linux is connected to a Linux PC through an Ethernet cable.

1. On the Linux PC, open a terminal and run the *ifconfig* command to find the IPV4 Address
   a. (169.254.132.246 in this example, labeled *inet*)
2. On the testboard, run *ifconfig 169.254.132.250* to assign a static IP address
   a. Note that the address only has the final three decimals changed to be unique
      i. Known as the host ID
   b. The rest of the address (169.254.132) must be the same
      i. Known as the network ID
3. From the testboard, run *ping 169.254.132.246*(the IP address of the Linux PC)
   a. Ping begins and can be stopped by pressing 'ctrl' and 'c' at the same time.

The codeblock is captured from the testboard pinging to the Linux PC, but ping can be performed both ways, for example, Linux PC can also ping the testboard.

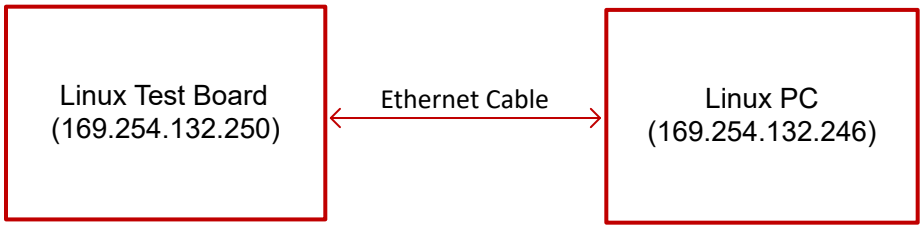

Figure 4-1. Ping Block Diagram Example

```
root@j7-evm:~# ifconfig eth0 169.254.132.250
root@j7-evm:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500  metric 1
        inet 169.254.132.250  netmask 255.255.0.0  broadcast 169.254.255.255
        inet6 fe80::3608:e1ff:fe59:5cd2  prefixlen 64  scopeid 0x20<link>
        ether 34:08:e1:59:5c:d2  txqueuelen 1000  (Ethernet)
        RX packets 133  bytes 16347 (15.9 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 481  bytes 117318 (114.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@j7-evm:~# ping 169.254.132.246
PING 169.254.132.246 (169.254.132.246): 56 data bytes
64 bytes from 169.254.132.246: seq=0 ttl=64 time=0.579 ms
64 bytes from 169.254.132.246: seq=1 ttl=64 time=0.546 ms
64 bytes from 169.254.132.246: seq=2 ttl=64 time=0.587 ms
64 bytes from 169.254.132.246: seq=3 ttl=64 time=0.557 ms
64 bytes from 169.254.132.246: seq=4 ttl=64 time=0.518 ms
64 bytes from 169.254.132.246: seq=5 ttl=64 time=0.574 ms
64 bytes from 169.254.132.246: seq=6 ttl=64 time=0.548 ms
64 bytes from 169.254.132.246: seq=7 ttl=64 time=0.561 ms
^C
--- 169.254.132.246 ping statistics ---
8 packets transmitted, 8 packets received, 0% packet loss
round-trip min/avg/max = 0.518/0.558/0.587 ms
root@j7-evm:~#
```

With ping successfully working, we can attempt to perform a throughput test using *iPerf*, an open-source tool used to measure network performance/bandwidth. *iPerf* needs to be installed on both machines (testboard and Linux PC) to function.

iPerf Example:

1. On the test board, run the command *iperf -s* to establish that the test board acts as the server.
2. On the Linux PC, run the command *iperf -c 169.254.132.250* (the IP address of the server), to establish that the Linux PC acts as a client and connects to the server.

The codeblock is captured from the testboard. Here we can see 1.09 GB of data successfully transferred and the Bandwidth is very close to the advertised speed of the network port (1000Mbps).

```
root@j7-evm:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 169.254.132.250 port 5001 connected with 169.254.132.246 port 37356
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  1.09 GBytes   933 Mbits/sec    //This step happens after the Linux PC connects
as a client
```

5 Summary

This application note provides a comprehensive overview of basic Linux PHY driver terminology, guiding users through the integration of a new Ethernet driver into a system and offering insights into common terminal commands for debugging purposes. From foundational concepts to practical implementation, this resource equips developers with the knowledge and tools necessary to navigate the complexities of Linux driver development efficiently.

6 References

- Texas Instruments, *ti-ethernet-software Github*.
- Ethtool, *Linux Manual Page*.
- iPerf, *Speed Test Tool*.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265